How to Control the Check Point Publish Flow in Terraform

Introduction

The Check Point provider can be used to automate security responses to threats, provision both physical and virtualized next-generation firewalls and automate routine Security Management configuration tasks, saving time and reducing configuration errors. With the Check Point provider, DevOps teams can automate their security and transform it into DevSecOps workflows.
By Design, Terraform can handle many connections in parallel to speed up deployments. Meaning that the execution of resources is not in the same order as they appear in the configuration files. It is possible to control this behaviour using the flag below:

```
terraform apply -parallelism=1
```

Assume that we must create an access rule in the Check Point security management server. Rules usually contain objects like source host, destination subnet, applications, etc. Before creating the access rule, the objects that are used in the rule should be created first. Otherwise, those objects cannot be referenced and consider unknown.

In order to control the flow of execution, Terraform creates a dependency graph. To view the dependencies, you can use the command "Terraform Graph".

```
terraform graph
```

The output of the command above shows the relationship (dependency) of all objects handled in the configuration files.

While Terraform dependency management is very powerful, in certain scenarios, we need to Use the `depends_on` meta-argument to handle hidden resource or module dependencies that Terraform cannot automatically infer. You only need to explicitly specify a dependency when a resource or module relies on another resource's behaviour but does not access any of that resource's data in its arguments.

In addition to the default Terraform behaviour, the following challenges can create challenges in handling the flow correctly.

CHECK POINT™
YOU DESERVE THE BEST SECURITY

Challenges

1. Terraform lack the support of native support for publishing the changes, post apply actions are handled out-of-band.

This behaviour is documented by most vendors. For check Point see the link below

https://registry.terraform.io/providers/CheckPointSW/checkpoint/latest/docs#publish

The handling of the publish module is via a script that is run out of bank after the terraform execution is done.

```
$ cd $GOPATH/src/github.com/terraform-providers/terraform-provider-checkpoint/commands/publish
$ go build publish.go
$ mv publish $GOPATH/src/github.com/terraform-providers/terraform-provider-checkpoint
$ terraform apply && publish
```

While Check Point has support for publishing changes in the module below, Terraform will not be able to handle the dependencies. Meaning, all object and rules must be created before attempting to run the publish module. While we can define dependencies manually as explained above, this is not a simple task when handling many resources.

However, it is possible to define dependencies based on modules. Modules are container (folders) that groups multiple resources together under one structure. Read more regarding modules in the link below:

https://developer.hashicorp.com/terraform/language/modules

Splitting the resources from the publish resource into two separate modules allows us to have one single dependency defined between all the resources created on the management server as one module, while keeping the publish resource in a separate module.

Assume I have the code below to create 20 host objects (made as a loop for simplification, usually done as separate resources):

```
# Create 20 host objects
resource "checkpoint_management_host" "tf-host-standard" {
       count = var.details_level_full ? 0 : 20
       name = "tf-host-${count.index + 1}"
       ipv4_address = "10.10.0.${count.index + 1}"
       ipv6_address = "2001:db8:1::${count.index + 1}"
       color = "red"
       comments = "Created via terraform"
       nat_settings = {
       auto_rule = false
       }
}

# Publish Management Session
resource "checkpoint_management_publish" "publish" {
}
```

The code above falls under the Terraform limitation of handling publish and the only way to make sure that the publish is done in sequence and only after all resources are created is not possible natively.

We will split the resources and publish into two modules with the following structure:

```
main.tf
providr.tf
variables.tf

Hosts_module
       --->hosts.tf
       --->provifer.tf
       --->variables.tf

publish_module
       --->publish.tf
       --->provider.tf
       --->variables.tf
```

Next, we will create dependency between the publish module and the resource (hosts_module) and the Publish resource (publish_module) in the main configuration file "main.tf":

```
module "resources" {
       source = "./hosts_module"
}

module "publish" {
       source = "./publish_module"
       depends_on = [module.resources]
}
```

Notice that we used depends_on inside the publish resource. This instructs Terraform to execute the publish module only after all resources modules has been executed.

Below, you can see the resources are executed in order:

```
module.resources.checkpoint_management_host.tf-host-standard[16]: Creation complete after 35s
[id=8a97d6e8-4e99-4cac-b804-5f5efe0afbbc]
module.resources.checkpoint_management_host.tf-host-standard[1]: Creation complete after 12s
[id=39b3a781-d8ce-4330-a25c-141ec335a580]
module.publish.checkpoint_management_publish.publish: Creating...
module.publish.checkpoint_management_publish.publish: Creation complete after 2s [id=publish-
8rpxk2hrtu]
```

Notice that the Destroy (removing resources) has the opposite execution order. If a resource depends on another, it is removed first before the resource we depend on is removed. This means on destroy, the publish module is executed first and then the resources are destroyed.

To workaround this behaviour, we can add a publish module to run before the resources are created and after the resource is published. We can modify the structure as follow:

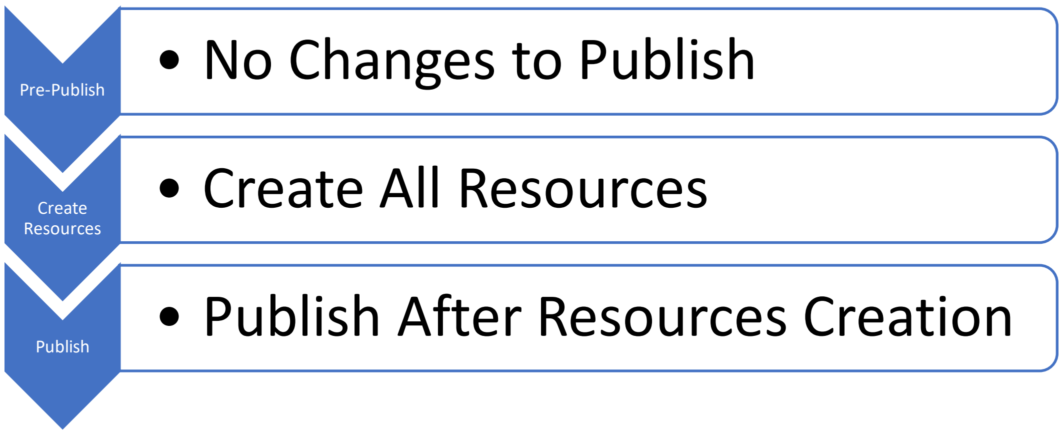
```
main.tf
providr.tf
variables.tf

pre_publish_module
       --->pre_publish.tf
       --->provider.tf
       --->variables.tf

Hosts_module
       --->hosts.tf
       --->provifer.tf
       --->variables.tf

publish_module
       --->publish.tf
       --->provider.tf
       --->variables.tf
```

Next, we will change the main.tf file and make the resources module depends on the pre_publish module.

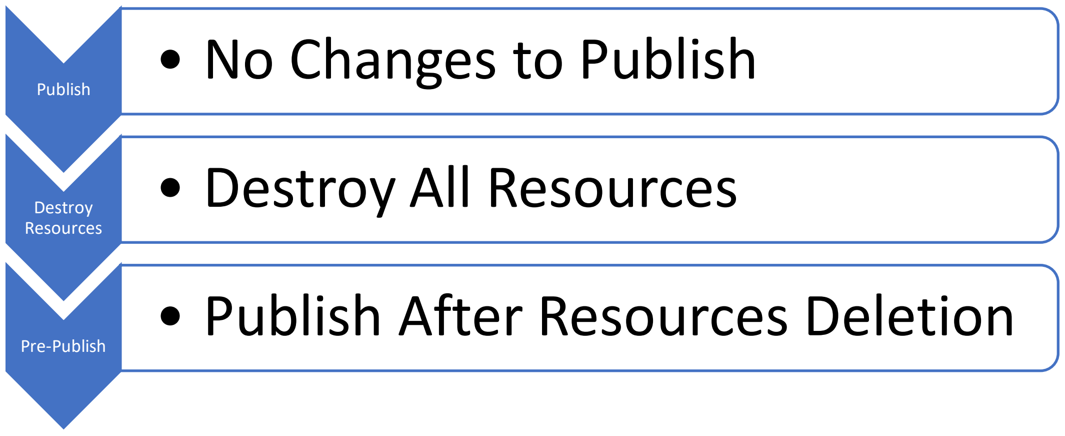
```
module "pre_publish" {
       source = "./pre_publish_module"
}

module "resources" {
       source = "./hosts_module"
       depends_on = [module.pre_publish]
}

module "publish" {
       source = "./publish_module"
       depends_on = [module.resources]
}
```

The flow of execution is as follow:

On Create:



On Destroy:



Notice: The publish resource does not get triggered on destroy by default. A new flag was introduced to allow this behaviour. See the example below:

```
resource "checkpoint_management_publish" "publish" {
       run_publish_on_destroy = true
}
```

2. Publishing many changes can take a long time and cause timeouts.

When publishing changes at the end, the Check Point management server should validate each resource, for example, validate that the object does not already exist, etc.

This can take a long time and cause a timeout when many changes are published only once at the end.

Check Point has introduced a new behaviour in the provider starting Version 2.5.1. a new flag has been introduced to allow the publish on intervals based on the number of changes. This flag can be used in the provider definition.

For example, you can use the code below to publish every 50 changes (50 is recommended):

```
provider "checkpoint" {
       server = "10.0.1.100"
       context = "web_api"
       #username = "admin"
       #password = "Cpwins!1"
       api_key = "eEvxQYF+4Y79K1297ezjQQ=="
       session_name = "TF Auto Publish"
       auto_publish_batch_size = 50
}
```

Using the code above, at every **50** changes, the terraform provider will publish the changes and then proceed to creating the next batch of resources.